A Seamless Services Migration Framework With JVM Tool Interface

Jia Dai
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
jia.dai@polito.it

Maurizio Morisio
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
maurizio.morisio@polito.it

Jose F. Mejia Bernal
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
jose.mejiabernal@polito.it

***Abstract*—Services migration is becoming more and more important due to the flexibility of the complex distributed systems. In the Java environment, the problem may not be solved without modification on JVM or bytecode in existing solutions. This paper presents a migration framework extended by IBM Service Management Framework; then it provides an experiment analysis and evaluation on a typical use case. Moreover, it shows a new approach for Java thread migration by using JVM Tool Interface in order to realize a transparent migration at a combination of application and thread level for enhancing capability of the framework. Our solution is implemented and evaluated to show the functionality of the prototype.**

***Index Terms*—Service Oriented Architecture, service migration, IBM Service Management Framework, JVM Tool Interface**

I. INTRODUCTION

Due to the rising interests in these service oriented architectures and flexibility of the complex distributed systems, a seamless and transparent service migration has become an important topic. The service migration can be used to balance the load between nodes [1], to reduce network traffic by moving clients closer to the accessed servers [2] or to cluster and scheduling [3]. Obviously, a good migration approach should enhance the performance of the SOA system.

In this context, the object paradigm has proven to be well suited to distributed applications development and the Java Virtual Machine (JVM) is now considered as a reference platform. JVM provides many services for distributed applications development. One of the most important aspects of Java (regarding distribution) is mobility. Java allows instances and code to be moved between machines. Java provides a serialization mechanism [4] which allows the capture and restoration of objects' states and therefore the migration of objects between machines. It also allows classes to be dynamically loaded and therefore to be moved between nodes. All these features lead to the development of mobile agent systems, whose main advantage is to provide agent migration between machines.

However, one mechanism is missing. Java does not provide any mechanism for capturing and restoring the thread state. The stack of Java threads is not accessible directly. That is, in order to capture the state of an application, Java only grants access to its objects and classes, the stack of the thread remaining inaccessible [14].

II. RELATED WORK

Many researchers focus on the migration topic. We intend to split the service migration problem into two parts, or levels: the process level and thread level. For the process level, it can be solved by using Java serialization mechanism. We focus on the thread level.

Several projects have addressed the issue of Java thread migration, such as Sumastra [5], Merpati [6], CIA [7], JESSICA2 [8], LAOVM [9], Wasp [10], Brakes [11], JavaGo [12] and M-DSA [13]. The problem is that they are based on a pre-processor that changes the code in order to add statements which capture and restore the state of thread before its execution. However, they cannot get the whole state of a Java thread, because the state is internal to the JVM. The migration results are incomplete [14].

To realize a seamless and transparent service migration, we propose our solution as follows:

First of all, we present our service migration framework, Mobile Services Framework (MoSeF) based on IBM Service Management Framework (SMF), to realize the migration on the process level. Then we present a generic approach to realize a transparent Java thread migration by implementing JVM Tool Interface for enhancing the MoSef framework's capability.

The paper is organized as follows: Section 3 explains the design of the framework analyzes a typical scenario and presents the results. Section 4 explains the approach of Java thread migration. Finally, section 5 presents conclusions and future work descriptions.

III. MOSEF - EXTENSION FRAMEWORK OF RE-LOCATABLE SERVICES

IBM Service Management Framework (SMF), an implementation of the OSGi Service Platform specification [15], provides for the network delivery and management of applications and services independent of operating system and

instruction set architecture (ISA). Our framework, called Mobile Services Framework (MoSeF), is an extension of SMF, which gives a solution of service migration with service's state.

A. *Architecture*

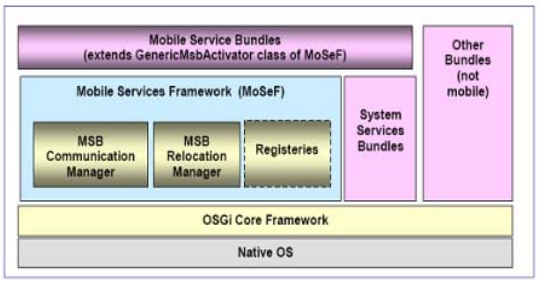

Figure 1. Architecture

MoSeF is built on SMF [16] and strengthens SMF by providing better availability and scalability to the services running on it. MoSeF is deployed as a service bundle within SMF environment. The Fig. 1 shows the logical component architecture of MoSeF.

A SMF bundle becomes a Mobile Service Bundle (MSB) by extending the GenericMsbActivator class, which is provided as a part of MoSeF implementation. This class has various callback methods that are intercepted by the MoSeF container hosted on the same SMF node and it takes control of the life-cycle of the MSB. Thus not only these bundles are able to use of the services provided by the MoSeF, but MoSeF can also act upon them.

Lifecycle of a MSB is a super set of the life cycle of a normal bundle, i.e. a MSB will have all the possible states that a SMF bundle has, but in addition it can also be in some other well defined states specific to MoSeF context.

There are two ways to create a new instance of a MSB. The first one is to instantiate a completely new MSB from class definitions/template available either locally or remotely. Another one is to create a clone of an existing MSB. That we use for migration.

The cloned MSB has the same state as the original one but has a distinct identity of its own. Once created, a MSB can be dispatched to and/or retracted from a remote location, deactivated and placed in secondary storage, then activated later. And this cycle may continue in forward or backward direction. MoSeF is the main component which brings relocatability as an add-on feature to the SMF bundles. It comprises of four sub-components, showed by Table 1.

TABLE I. MoSeF componets' functions

MSB Relocation Manager	Handles the stateful relocation and clones MSBs across the MoSeF node
MSB Communication Manger	Enables communication between ● MSB instances ● MSBs and Registries
Host Registry	Maintains a record of the currently active MoSeF nodes. The following operations are supported by Host Registry: ● Register: to register a MoSeF node ● Unregister: to unregister a MoSeF node ● Query: to query for a matching MoSeF node for a required node profile (properties like CPU, Memory) ● Update Property: to update the properties associated with a registered MoSeF node
Service Registry	Hosts a record of all the services and their location in the MoSeF domain

B. *Typical Scenario*

Location transparency is a key feature by which client applications are kept transparent of service migration activities. We will demonstrate it with a sample. There are two parts in the sample: a service provider MSB and a client to this MSB.

In this sample, client accesses the service from the IP obtained from the Service Registry and submits job to the MSB. After the job is completed the client can obtain the result from the MSB by accessing it from the same URI, as shown in Fig. 2.

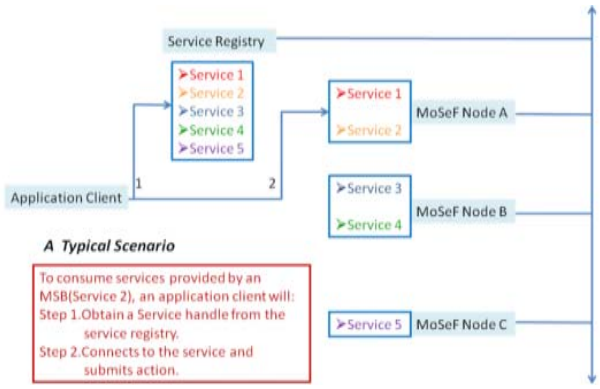

Figure 2. Scenario of no Service Migration in MoSeF

If in the meantime the MSB has moved to another location then the Client faults and again accesses the Service Registry to obtain the new location of the MSB. Since the MSB moves with all its state it will still have the results of the job submitted by this client. Now the client accesses the MSB from the new location and obtains the result of the job submitted earlier. All of them are done transparently to the user, as shown in Fig.3.

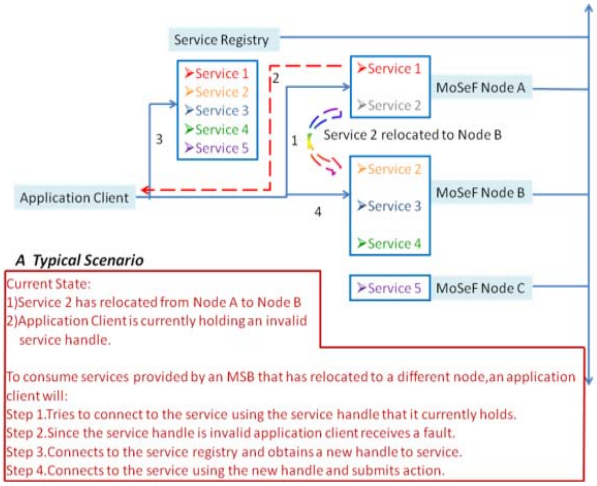

Figure 3. Scenario of Service Migration in MoSeF

C. Case Study

Let's take a distributed factorial calculation application for example. After installing the calculation service MSB in our framework, we could start running the client from another PC. It will launch a simple GUI that is used to transfer the numbers to the server, receive the result and show it, as shown in Fig.4.

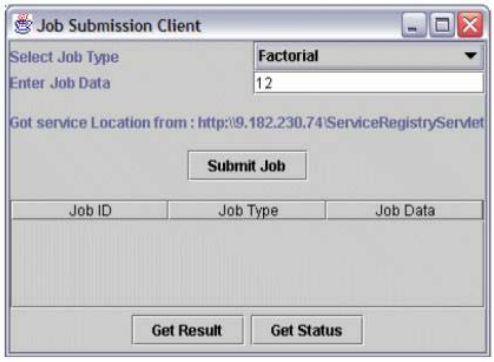

Figure 4. Interface of Factorial Calculation Client API

In order to simulate the migration, we move the service from one server to another. And we find that each movement of service will cause the source node to generate a new .jar file under the working directory of SMF, and the destination node will also receive the same .jar file under the working SMF directory, demonstrating the service name and the source node's IP address. Unzip this .jar file, we will find out it includes the following files: two serialized byte stream files, ended with .ser, and the source code file of the service. By using Java deserialization program, we can understand that each of them includes an object of hash table that stores all the requests from client, and also some user interface objects in corresponding with the special interface of this application program.

MoSeF in fact realizes the stateful relocation of service instances in an upper level. It packs the service class files with the relative request variables and then moves the package among nodes when interruption occurs during service execution. And with the help of SMF, it can be reinstalled and restarted on the new destination nodes. The process is transparent to the client, because the service updates all the information by itself after migration. So in the client's point of view, there are no changes about the service.

However, when we move the running service, problems occur. We cannot get result after the movement. It shows that, although the service is well migrated and restarted successfully on the new node, the interrupted calculation thread cannot be resumed because of the lack of the running context. We need a new approach for Java thread migration.

IV. JAVA THREAD MIGRATION

As we mentioned before, Java virtual machine (JVM) does not allow threads to be migrated directly. To solve the problem, firstly, we capture the execution context and source code. Then we reproduce the context at the destination. At the end, we resume the execution of the thread.

Based on the analysis of JVM's structure and mechanism of Java thread, we conclude that the thread's state is fully stored in the Java Method Frame (JMF), and as if we can froze this state and transport it together with the service source code and the service state, we can recover not only the service and the client request, but also the thread. Then the thread continues its execution at the destination node.

The JVM Tool Interface (JVMTI) is a programming interface used by development and monitoring tools. It provides both methods to inspect the state and control the execution of applications running on the JVM. A client of JVMTI, hereafter called an agent, can be notified of interesting occurrences through events. JVMTI can query and control the application through many functions, either in response to events or independent of them. Agents run in the same process and communicate directly with the virtual machine. This communication is through a native interface (JVMTI). The native in-process interface allows maximal control with minimal intrusion on the part of a tool. Typically, agents are relatively compact. They can be controlled by a separate process which implements the bulk of a tool's functions without interfering with the target application's normal execution.

A. Agent Initialization

The agent must contain a function called *Agent_OnLoad*, which is invoked when the library is loaded. The *Agent_OnLoad* function is used to set up functionality that is required prior to initializing the JVM. We must enable several capabilities for the JVMTI functions and events that we will use later.

It is generally desired, and in some cases required, to add these capabilities in the *Agent_OnLoad* function. For example, to use the *InterruptThread* function, the *can_signal_thread* capability must be true. In addition, the *Agent_OnLoad* function is often used to register for notification of events. Each event for which we register must also have a designated callback function, which will be called when the event occurs.

For example, if a JVMTI Event of Exception occurs, our example agent sends it to the callback method, *callbackException()*. The jvmtiEventCallbacks structure and *SetEventCallbacks* function will be used:

```
jvmtiEventCallbacks callbacks;
(void)memset(&callbacks, 0, sizeof(callbacks));
callbacks.VMInit = &callbackVMInit; /* JVMTI_EVENT_VM_INIT */
callbacks.VMDeath = &callbackVMDeath; /* JVMTI_EVENT_VM_DEATH */
callbacks.Exception = &callbackException;/* JVMTI_EVENT_EXCEPTION */
callbacks.VMObjectAlloc = &callbackVMObjectAlloc;/* JVMTI_EVENT_VM_OBJECT_ALLOC */
error = (*jvmti)->SetEventCallbacks(jvmti, &callbacks,(jint)sizeof(callbacks));
check_jvmti_error(jvmti, error, "Cannot set jvmti callbacks");
```

In the Agent_OnLoad function, we perform the following setup:

```
static GlobalAgentData data;
(void)memset((void*)&data, 0, sizeof(data));
gdata = &data;
...
/* Here we save the jvmtiEnv* for Agent_OnUnload(). */
gdata->jvmti = jvmti;
```

We create a raw monitor in Agent_OnLoad(), then wrap the code of VM_INIT, VM_DEATH and EXCEPTION with JVMTI RawMonitorEnter() and RawMonitorExit() interfaces.

B. Analyzing Threads By JVMTI

As discussed before, when the JVM starts, the startup function *Agent_OnLoad* in the JVMTI agent library is invoked. During the JVM initialization, a JVMTI Event of type *JVMTI_EVENT_VM_INIT* is generated and sent to the *callbackVMInit* routine in our agent. Once the JVM initialization event is received (that is, the VMInit callback is invoked), the agent can complete its initialization. Now, the agent is free to call any Java Native Interface (JNI) or JVMTI function.

After that, we enable the Exception events (*JVMTI_EVENT_EXCEPTION*) in the VMInit callback routine. We can get information about the monitors owned by the specified thread by using the JVMTI method: *GetOwnedMonitorInfo*. This function does not require the thread to be suspended. We can also get state information for a thread by using the JVMTI method: *GetThreadState*.

C. Obtaining a JVM Thread Stack Trace

The JVMTI method *GetStackTrace* can be used to get information about the stack of the thread. If max_count is less than the depth of the stack, the number of deepest frames is returned, otherwise the entire stack is returned. The thread is not suspended when we invoke the method.

The JVM Object Allocation event is useful for determining information about objects allocated by the JVM. In the *Agent_OnLoad* method, we registered *callbackVMObjectAlloc* as the function to be called when the JVM Object Allocation event is sent. The parameters of callback method contain the information about the objects that have been allocated, such as the JNI local reference to the class of the object and the object size. With the jclass parameter, object_class, we can use the *GetClassSignature* method to obtain the information about the names of the classes.

We use the *GetStackTrace* method to print the stack trace of the threads that is allocating the object. As that section describes, we obtain frames with a specified depth. The frames are returned as *jvmtiFrameInfo* structures, which contain each frame's *jmethodID* (that is, frames[x].method). The *GetMethodName* method can map the *jmethodID* to that particular method's name. Finally, we use the *GetMethodDeclaringClass* and *GetClassSignature* methods to obtain the names of the classes from which the method was invoked.

D. Analyzing the Heap Using JVMTI

In addition, some iteration methods in JVMTI allow you to iterate over the entire heap (both reachable and unreachable objects). That means we will cover the root objects and all objects that are directly and indirectly reachable from the root objects, and all objects in the heap that are instances of a specified class. During the execution of these functions, the state of the heap is not changed: no objects are allocated, no objects are garbage collected, and the state of objects (including held values) is not changed.

Generally speaking, we use the related functions in JVMTI in order to control the threads' execution. When threads are migrated, we use JVMTI to suspend threads firstly, analyze the running states and then serialize them. After finishing the service migration, we reproduce the running context and resume the threads by JVMTI.

V. CONCLUSION AND FUTURE WORK

MoSeF is an extension for SMF, which aims to provide a seamless service migration. With the support of the SMF, MoSeF succeeds in serializing the services' state, packaging it with services' source files, transporting and restoring the service at the destination node. However, it cannot recover the threads' execution state and restart the already executing threads at the destination node. In this case, MoSeF can hardly provide uninterrupted user experience.

To solve this, our solution is that move threads' states together with service and restart the threads corresponding to certain client's request at the destination node. The solution is a finer-granularity service migration and would provide more reliable and transparent service on service oriented architecture. Since JVMTI provides both ways to inspect the state and control the execution of applications running in JVM, it is a possible way to realize thread migration in our solution. We make no changes to JVM, no modification on bytecodes and transmit the complete running state.

We implemented and evaluated the functionality of our prototype by applying it to thread migration and thread persistency. The future work is to compare the performance, effects and cost of different levels of service migration.

REFERENCES

[1]. D.A. Nichols. Using Idle Workstations in a Shared Computing Environment. Proceedings of the 11th ACM Symposium on Operating Systems Principles, pages 5-12, ACM 8-11, November 1987.

[2]. F. Douglis et B. Marsh. The Workstation as a Waystation: Integrating Mobility into Computing Environments. The 3rd Workshop on Workstation Operating System (IEEE), april 1992.

[3]. D. Chess, C. Harrison et A. Kershenbaum. Mobile Agents: Are They a Good Idea?. IBM ResearchReport. IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, march 1995.http://www.research.ibm.com/iagents/publications.html

[4]. R. Riggs, J. Waldo, A. Wollrath, K. Bharat. *Pickling State in the Java System*. USENIX Conference on Object-Oriented Technologies (COOTS), Ontario, Canada, 1996.

[5]. Acharya, A., Ranganathan, M., and Salz, J., Sumatra: A Language for Resource-aware Mobile Programs, Mobile Object Systems: Towards the Programmable Internet, Lecture Notes in Computer Science, Number 1222, April 1997.

[6]. Suezawa,T., Persistent Execution State of a Java Virtual Machine, Proceedings of the ACM 2000 Java Grande Conference, San Francisco,

California, USA,June 2000

[7]. Illmann, T., Krueger, T., Kargl F., Weber, M., Transparent Migration of Mobile Agents Using the Java Debugger Architecture, Proceeding of the Fifth IEEE International Conference on Mobile Agents (MA 2001), Atlanta, Georgia, USA, December 2 - 4, 2001.

[8]. W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, September 2002.

[9]. X. Liao, Y. Yue, H. Jin and H. Liu. LAOVM: Lightweight Application-Oriented Virtual Machine for Thread Migration. icis, pp.882-887, 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science (icis 2009), 2009

[10]. Funfrocken, S., Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs), Proceeding of Second International Workshop Mobile Agents 98 (MA 98), Stuttgart, Germany, September 9 – 11, 1998.

[11]. Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., and Verbaeten, P., Portable Support for Transparent Thread Migration in Java, Proceeding of the the Fourth International Symposium on Mobile Agents 2000 (MA 2000), Zurich, Switzerland, September 13 – 15, 2000.

[12]. Sakamoto, T., Sekiguchi, T., and Yonezawa, A., Bytecode Transformation for Portable Thread Migration in Java, Proceeding of the Fourth International Symposium on Mobile Agents 2000 (MA 2000), Zurich, Switzerland, September 13 – 15, 2000.

[13]. S. Fu and C.-Z. Xu. Service migration in distributed virtual machines for adaptive grid computing. Technical report, Department of Electrical and Computer Engineering, Wayne State University, March 2004.

[14]. Sara Bouchenak and Daniel Hagimont, Zero Overhead Java Thread Migration, Technical Report 0261, INRIA, 2002.

[15]. The Open Service Gateway initiative (OSGi) website. http://www.OSGi.org

[16]. The IBM Service Management Framework is an implementation of OSGi Service Platform specification that provides network delivery and management of services. Homepage: http://www-306.ibm.com/software/wireless/smf

n